2

NASA Contractor Report 187514
ICASE Report No. 91-12

ICASE

PERFORMANCE EFFECTS OF IRREGULAR COMMUNICATIONS PATTERNS ON MASSIVELY PARALLEL MULTIPROCESSORS

Joel Saltz
Serge Petiton
Harry Berryman
Adam Rifkin

Contract No. NAS1-18605
January 1991

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

NASA
National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23665-5225

Performance Effects of Irregular Communications Patterns on Massively Parallel Multiprocessors

Joel Saltz*
ICASE, NASA Langley Research Center
Hampton, VA 23665

Serge Petiton
Etablissement Technique Central de l'Armement
16 bis Avenue Prieur de la cote d'Or
94114 Arcueil Cedex France

Harry Berryman
ICASE, NASA Langley Research Center
Hampton, VA 23665

Adam Rifkin
Computer Science Department
College of William and Mary
Williamsburg, VA 23185

ABSTRACT

We conduct a detailed study of the performance effects of irregular communications patterns on the CM-2. We characterize the communications capabilities of the CM-2 under a variety of controlled conditions.

In the process of carrying out our performance evaluation, we develop and make extensive use of a parameterized synthetic mesh. In addition we carry out timings with unstructured meshes generated for aerodynamic codes and a set of sparse matrices with banded patterns of non-zeros. This benchmarking suite stresses the communications capabilities of the CM-2 in a range of different ways. Our benchmark results demonstrate that it is possible to make effective use of much of the massive concurrency available in the communications network.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665, and by NSF Grants ASC-8819373 and ASC-8819374.

1 Introduction

Many algorithms used to solve extremely large computationally intensive problems are characterized by irregular patterns of data access. It is often possible to substantially reduce memory requirements in computations that arise in linear algebra by using *sparse* algorithms [8], [20]. Sparse algorithms avoid explicitly storing matrix elements whose values are known to be 0. Sparse algorithms are particularly important in large computationally intensive problems since computationally intensive problems frequently require large amounts of computer memory. Sparse linear algebra algorithms are employed to solve problems from a variety of areas including structural analysis and oil reservior simulations. Irregular data structure access patterns also occur in unstructured mesh methods used to solve problems in computational fluid dynamics [13], [25]. The data structures in sparse algorithms and in unstructured mesh methods tend to get referenced in non-uniform ways.

We carry out a detailed study of the performance effects of irregular communications patterns on the CM-2 by studying iterative sparse matrix vector multiplication. Sparse matrix vector multiplication is one of the simplest possible sparse computational kernels. Sparse matrix vector multiplication is a basic component of many iterative methods that arise in linear algebra. Examples of such algorithms include Krylov subspace linear equation solvers [23], [20] as well as subspace method eigenvalue solvers.

Two different algorithms to carry out sparse matrix vector multiplication are defined, coded and benchmarked. The performances of these algorithms are critically dependent on the sparse matrix employed and on the way in which the sparse matrix is mapped to the machine. To benchmark these sparse matrix vector multiplication methods, we develop and employ a parameterized synthetic mesh, unstructured meshes generated for aerodynamic codes, and a set of sparse matrices with banded patterns of non-zeros. This benchmarking suite stresses the communications capabilities of the CM-2 in a range of different ways.

Most of our experiments involve use of a communications routing algorithm that schedules relatively high bandwidth concurrent communications simultaneously over all communications channels ([6]). Several modern multiprocessors are capable of supporting the simultaneous use of all communications channels, [4], [19]. Consequently, the results described here should shed light on the considerations that go into solving irregular problems on a variety of multiprocessor architectures.

In Section 2 we describe the methods we use to carry out massively parallel sparse matrix vector multiplication, in Section 3 we describe the test matrices we use in our CM-2 experiments. We describe the CM-2 communications architecture in Section 4 and describe how we map problems onto the processors of the CM-2 in Section 5. We present our experimental data in Section 6. This data outlines the performance characteristics of the CM-2 communication links, the communication compiler and the CM-2 general router. We also explore the performance impact of the geometrically based mapping of problems. Finally we compare the performance achieved by the two parallel sparse matrix vector multiply methods proposed in Section 2.

2 Massively Parallel Sparse Matrix Vector Multiply Variants

In this section, we outline a number of different ways one can carry out sparse matrix vector multiplications on massively parallel machines.

2.1 Notation

Assume that A is an N by N matrix. We will assume that the matrix has a maximum of C_R non-zero entries in each row and a maximum of C_C non-zero entries in each column. There are a wealth of notations and format representations that have been used to store sparse matrices [20]. We use a representation in this paper that closely corresponds to the formats we will use for storing data in massively parallel machines.

We will use the matrix in Figure 1 to illustrate our matrix representations. We introduce the *Jagged Diagonal Row* storage format [24]. The non-zero elements of A are stored in a size N by C_R matrix a. A size N by C_R matrix ia stores the columns of A in which non-zero elements appear. The entry a(i,j) represents the jth non-zero entry in row i of matrix A. The entry ja(i,j) represents the column in A of the jth non-zero entry in row i of A. We depict in Figure 2, the matrices a and ja that correspond to the example in Figure 1.

We will map various axes of our sparse data arrays onto multiprocessor arrays. We will assume that the number of processors in a processor array axis is greater than or equal to the number of matrix entries in the corresponding sparse data array axis. This is a convenient assumption to make in programming environments that allow each physical processor to simulate varying numbers of *virtual processors.*

If a one dimensional N element array x is spread out among N processors, each processor is able to store its value of x as a scalar variable. If we map the first dimension of a size N by M matrix b onto N processors, each processor will store a column of b in a size M one dimensional local array. If we map b onto a N by M two dimensional processor array, each processor will store a single element $b(i,j)$ of b.

2.2 One Row per Virtual Processor Algorithm

We will outline a row oriented algorithm that carry out matrix vector multiplies using N virtual processors. In this discussion we will assume that the N virtual processors are configured in a linear array. We map the right hand side vector x and the solution vector y so that $x(i)$ and $y(i)$ are placed on processor i in the virtual processor array. We also map the columns of a and ja onto the virtual processor array so that rows i of a and ja are assigned to processor i. These array mappings are depicted in Figure 3. Each processor i obtains the elements $x(ja(i,j))$ from processors $ja(i,j)$, for $1 \leq j \leq C_R$. These array elements are stored on processor i as depicted in Figure 4. Each processor i then concurrently carries out the necessary inner product and places the result in y (Figure 5).

Figure 1: Array for illustrating data format

$$\begin{array}{cccc} A_{1,1} & 0 & A_{1,2} & 0 \\ 0 & A_{2,2} & 0 & A_{4,2} \\ 0 & A_{3,2} & A_{3,3} & 0 \\ A_{4,1} & 0 & 0 & A_{4,4} \end{array}$$

Figure 2: Matrix a and ja

$$\begin{array}{cc} A_{1,1} & A_{1,2} \\ A_{2,2} & A_{4,2} \\ A_{3,2} & A_{3,3} \\ A_{4,1} & A_{4,4} \end{array} \qquad\qquad \begin{array}{cc} 1 & 3 \\ 2 & 4 \\ 2 & 3 \\ 1 & 4 \end{array}$$

Figure 3: One Row per Virtual Processor Data Placement

Proc 1	$x(1)$	$y(1)$	$a(1,1)$	...	$a(1,C_R)$	$ja(1,1)$	...	$ja(1,C_R)$
Proc 2	$x(2)$	$y(2)$	$a(2,1)$	...	$a(2,C_R)$	$ja(2,1)$	...	$ja(2,C_R)$
...	...	...	...	...	...	...	...	...
Proc N	$x(N)$	$y(N)$	$a(N,1)$	...	$a(N,C_R)$	$ja(N,1)$	...	$ja(N,C_R)$

Figure 4: Storage of x after Communication

Proc 1	$x(ja(1,1))$	...	$x(ja(1,C_R))$
Proc 2	$x(ja(2,1))$	...	$x(ja(2,C_R))$
...	...	...	...
Proc N	$x(ja(N,1))$	...	$x(ja(N,C_R))$

Figure 5: Matrix Vector Multiply Inner Products

Proc 1	$y(1) = \sum_{j=1}^{C_R} a(1,j) * x(ja(1,j))$
Proc 2	$y(2) = \sum_{j=1}^{C_R} a(2,j) * x(ja(2,j))$
...	...
Proc N	$y(N) = \sum_{j=1}^{C_R} a(N,j) * x(ja(N,j))$

Figure 6: Reduction Algorithm - Placement of arrays a and x in N by C_R Processor Array

$$\begin{array}{cccc}
a(1,1) & a(1,2) & \ldots & a(1,C_R) \\
a(2,1) & a(2,2) & \ldots & a(2,C_R) \\
\ldots & \ldots & \ldots & \ldots \\
a(N,1) & a(N,2) & \ldots & a(N,C_R)
\end{array}
\qquad
\begin{array}{cccc}
x(1) & 0 & \ldots & 0 \\
x(2) & 0 & \ldots & 0 \\
\ldots & \ldots & \ldots & \ldots \\
x(N) & 0 & \ldots & 0
\end{array}$$

Figure 7: Reduction Algorithm - Assignment of x after Communication

$$\begin{array}{ccc}
x(ja(1,1)) & \ldots & x(ja(1,C_R)) \\
x(ja(2,1)) & \ldots & x(ja(2,C_R)) \\
\ldots & \ldots & \ldots \\
x(ja(N,1)) & \ldots & x(ja(N,C_R))
\end{array}$$

2.3 Reduction Algorithm

We next present an algorithm that uses reductions to compute the sparse matrix vector multiply sums. This algorithm takes place on a size N by C_R array of virtual processors. Elements $a(i,j)$ and $ja(i,j)$ are assigned to virtual processor (i,j), and elements of the right hand side vector x are assigned to the *first column* of virtual processors (Figure 6). A communication phase follows in which $x(ja(i,j))$ is placed on virtual processor (i,j) (Figure 7). The multiplication of $a(i,ja(i,j))$ by $x(ja(i,j))$ is carried out on processor (i,j). Once the products $a(i,ja(i,j))*x(ja(i,j))$ are computed, a reduction along each row of the virtual processor array can be used to perform the sums $\sum_j^{C_R} a(i,ja(i,j))*x(ja(i,j))$. The resulting solution vector is stored in the first column of the processor grid.

2.4 Reduction Algorithm Variants

There are many different variants of reduction algorithms.

There is a useful variant of the reduction algorithm described in Section 2.3 in which we begin with copies of x(i) mapped to all processors in row i of the processor grid. This variant generates a one-to-one pattern of inter-processor communication. When a matrix-vector multiplication comprises a portion of a larger massively parallel program, it may be advantageous to begin carrying out the matrix vector multiplication with the right hand side vector layed out in this fashion. After the matrix vector multiply is carried out, each element of processor array row i ends up with a copy of the solution value. Such a layout has been used on in the implementation of iterative subspace methods on the CM-2 [16].

In another version of a reduction algorithm, *all* products are summed using a *single* segmented scan operation. In a segmented scan operation, one generates a running sum that restarts at certain given points. A sparse matrix vector multiply algorithm of this sort was described by Kumar [14].

2.5 CM-2 Reduction Algorithm Implementation

Actual multiprocessors have a specific number of processors; in many cases the number of real processors is substantially smaller than the number of non-zero elements in a matrix. In that case, some portion of the algorithm will be sequentialized. We use the *create_detailed_geometry* PARIS instruction ([15]) in the Reduction algorithm to ensure that the virtual processor dimension over which the reduction takes place is mapped to low order address bits. Thus the reduction is sequentialized when the problem is large enough to fit a row onto each physical processor.

3 Test Matrices

On a massively parallel architecture, matrix vector multiplication can be performed in a large number of different ways. We use a range of matrices to allow us to elucidate the resulting differences in performance. We use sparse matrices generated from finite difference templates, banded matrices and sparse matrices obtained from unstructured meshes used in aerodynamic simulations. We also generate parameterized synthetic meshes by generating random numbers and using these to incrementally distort matrices obtained from a fixed finite difference template. A detailed description of the matrices we use follows.

Banded Matrices

We generate banded matrices with size C bands. These matrices have size C bands distributed around the diagonal, with $C/2 - 1$ bands lying immediately below the diagonal and $C/2$ bands lying above the diagonal. Note that we do not use data structures that take advantage of the banded structure of these matrices.

Synthetic Workload from Templates

A finite difference template is used to link K points in a square two dimensional mesh. This connectivity pattern is incrementally distorted. Random edges are introduced subject to the constraint that in the new mesh, each point still requires information from K other mesh points.

This mesh generator makes the following assumptions:

1. The problem domain consists of a 2-dimensional square mesh of N points which are numbered using their row major or natural ordering,
2. Each point is initially connected to K neighbors determined by a finite difference template,
3. A matrix of pointers `ia` can be used to express the connectivity pattern of the mesh. We use Jagged Diagonal Row storage format.
4. With probability q , each $ja(i,j)$ will be replaced by a random integer between 1 and N.

In this paper we will use make use of two templates. One template connects each point to its four nearest neighbors (K=4). The other template connects each point to both its four nearest neighbors as well as to each of its four diagonal neighbors (K=8).

Unstructured Meshes from Aerodynamics

We use two different unstructured meshes generated from aerodynamic simulations.

> **Mesh A**: A 21,672 element mesh generated to carry out an aerodynamic simulation involving a multielement airfoil in a landing configuration [13]. This mesh has 11143 points.
>
> **Mesh B**: A 37,741 element mesh generated to simulate a 4.2% circular arc airfoil in a channel [11]. This mesh has 19155 points.

These unstructured meshes are used to investigate the importance of mapping in matrix vector multiplication in a realistic context. The ordering of rows in matrices derived from these unstructured meshes is not particularly regular and bears little relation to the kinds of natural row or column-wise orderings seen in matrices derived from structured meshes.

4 CM-2 Communications Architecture

A p processor CM-2 may be regarded as a $log_2(p) - 4$ dimensional hypercube in which each edge of the hypercube *is identified with a CM chips*. Each pair of CM chips is connected by a single bit wide data path. A p processor CM-2 has $log_2(p) - 5$ Weitek floating point accelerators. Two CM chips are associated with each Weitek chip, a *sprint* chip is used to provide the needed interface between the CM chips and the Weitek accelerators. It proves to be convenient to think of the CM-2 as a $log_2(p) - 5$ dimensional hypercube each node of which contains two CM chips, a sprint chip and a Weitek chip. In the rest of this paper, we will refer to this collection of four chips as a *sprint node*. It should be noted that each pair of CM-2 nodes are connected by a *pair* of single bit wide data paths.

When we program using the PARIS parallel instruction set, [15] data in the 32 processors associated with each CM-2 node is stored in bit-serial format. Because the floating point processors require data to be in 32 bit word-oriented format, the coupling between the bit serial processors and the floating point chip requires a data transposition. Thus, even though all floating point computation is carried out by the 32 bit floating point processors, the memory of each sprint node is fragmented into 32 separate segments.

A PARIS program is written in a manner that assumes the existence of an unlimited number of virtual processors. Interaction between virtual processors is carried out by passing messages. When a pair of communicating virtual processors are assigned to different sprint nodes, messages must traverse the intervening links. When communicating virtual processors are assigned to the same node, no communication along hypercube links is required.

The communication compiler [6] is a set of procedures used to schedule irregular communication patterns on the CM-2. Each processor calls the communication compiler scheduler and lists the processors with which communication is to take place. The communications compiler produces a schedule which is used to determine how messages will be routed through the hypercube channels. The architecture is able to make use of all hypercube links simultaneously. In a single router cycle, the CM-2 system software is able to carry out a bidirectional message transfer along each of the two links between each sprint node.

Messages are assigned to wires independently at each sprint node. An *assignment graph* is used to match messages with outgoing hypercube links. For each communication cycle, this assignment processes corresponds to picking links from a graph with two disjoint sets of nodes (i.e., a bipartite graph). The first set of assignment graph vertices represent messages that either

1. originate in the sprint node,
2. originate elsewhere and must be forwarded to their ultimate destination.

These vertices are called *message vertices.* The second set of assignment graph vertices consist of the $log_2(p) - 5$ *pairs* of hypercube links associated with each sprint node. These vertices are called *hypercube link vertices.* Constraints on this assignment problem are:

1. when a message has been assigned to a hypercube link, it cannot be moved over any other hypercube link during a given cycle,
2. once a hypercube link has been assigned to a message, it cannot transmit any other message during a given cycle, and
3. hypercube link assignments always decrease the hamming distance to a message's destination.

An a-edge is drawn from a message vertex M to a link vertex L when we route the message represented by M over the hypercube link represented by L. The communication compiler uses a heuristic assignment algorithm that attempts to maximize the number of messages sent during each communications cycle [6]. The degree ρ of an a-edge E of the assignment graph is defined as the sum of the number of a-edges leading out from E's message vertex and the number of a-edges leading out from E's hypercube link vertex. The algorithm begins by computing ρ for each a-edge. The a-edge S in the assignment graph with the smallest value of ρ is chosen. All a-edges that terminate on S's message or link vertex are removed and the process is repeated until all a-edges in the assignment graph have been chosen or eliminated.

5 Mapping on the CM-2

There are a wide range of sparse matrix problems in which a-priori mapping information can be derived. Many matrices arise from meshes. Mesh points that are physically close to one another are much more likely to be connected than more distant mesh points. Domain based geometrical information can be used to partition problems. The information can be propagated and used to map a sparse matrix onto a massively parallel machine. We can define mapped variants of the algorithms described in Sections 2.2 and 2.3.

One method we use to quantify the importance of judiciously mapping a domain onto processors is to map the synthetic mesh from templates in the following ways:

- map onto a 2-D set of virtual processors in a way that minimizes the number of hypercube links that must be traversed while communicating, or
- map onto a 1-D set of virtual processors.

The CM-2 software embeds both the 1-D and 2-D sets of virtual processors into the hypercube using a binary reflected Gray code. When we map the synthetic mesh onto a 1-D set of virtual processors, we guarantee non-local patterns of patterns of communication. By comparing results from the 1-D and 2-D mappings, we obtain an idea of the performance differences we can expect between

systematically partitioning and mapping a matrix to reduce communication requirements versus

embedding consecutively numbered rows of a matrix derived from a naturally ordered finite difference mesh into consecutively numbered processors in a 1-D VP set.

We can make use of geometric information when we partition matrices that arise from unstructured meshes (Section 3). In these cases, each mesh point is associated with a point in a spatial domain. The spatial domain can be partitioned in an attempt to minimize communications requirements while maintaining a balance of load. One of the commonly used partitioning methods is binary dissection [1]. The decomposition produced by binary dissection can be embedded into a hypercube. We can define *Jagged partitioning*, a simpler partitioning scheme that produces a decomposition that can be Gray code embedded into a two dimensional processor grid in a particularly straightforward manner us. The motivation for this partitioning is to produce a roughly uniform communications flux over the two hypercube dimensions involved in the two dimensional Gray coded mesh. We represent the number of neighbors linked to mesh point p by $N(p)$. We estimate the work associated with a subset S of the domain as

$$W(S) = \sum_{p \epsilon S} N(p).$$

Assume we want to create an M by N domain partitioning,

1. partition domain into M vertical strips S_i, so that the $W(S_i)$ values are equal and
2. partition each S_i into N chunks $C_{i,j}$ so that the $W(C_{i,j})$ values are equal.

Figure 8 depicts the results of this algorithm when applied to a illustrative domain. In this example, we assume that we have a very fine elliptically shaped mesh in the center of an otherwise unmeshed domain.

We used both schemes to partition the test aerodynamic grids, both schemes generated a comparable volume of interprocessor communication. Because we partitioned our synthetic meshes in a rectilinear manner and mapped the resulting matrices onto a two dimensional Gray coded mesh, for the sake of consistency, we chose the Jagged partitioning. It should be noted that both partitioning methods were mapped to the Intel iPSC/860 and very little performance distinction was found between the two partitioning schemes [21]. A performance comparison on the CM-2 remains to be carried out.

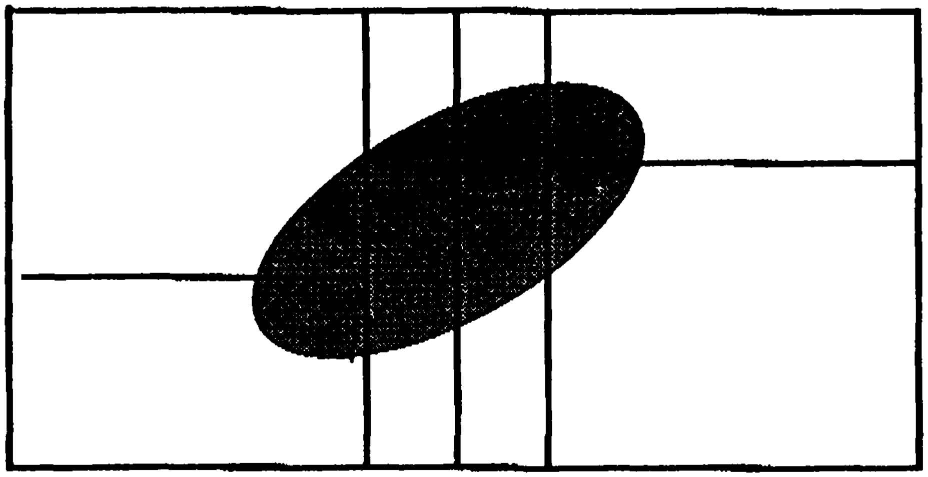

Figure 8: Jagged Partitioning of Unstructured Mesh for Multielement Airfoil

6 Experimental Data

In the following subsections, we will describe the results of the following computational experiments. These experiments were carried out on a 32K processor CM-2 located at NASA Ames Research Center, on a 8K processor CM-2 located at Yale University and on a 32K processor CM-2 located at Etablissement Technique Central de l'Armement (France).

In Section 6.1, we study the performance characteristics of the CM-2 communications links and of the communication compiler.

In Section 6.2, we carry out a performance comparison between the NEWS network, the CM general router, and the communications compiler on matrices formed from the synthetic mesh (Section 3). These matrices are mapped to the CM-2 using domain based geometric information.

In Section 6.3, we use the synthetic mesh and the unstructured meshes from aerodynamics to explore the performance impact of geometrically based problem mapping. The computational experiments use the communications compiler.

In Section 6.4 we compare the performance achieved by the massively parallel sparse matrix vector multiply variants described in Section 2.

In all but Section 6.4, we will employ the One Row per Virtual Processor algorithm.

6.1 Communications Compiler Performance

In this experiment, the communications compiler was used to schedule communications required for processor A to fetch from the memory of processor B. In Table 1, we present the time needed for a given processor A to fetch from the memory of processor 0. With the exception of processor A, all other processors fetch from their own memory. Each fetch was carried out 10,000 times, the average fetch time was calculated. The low order 5 binary bits of a processors address determine its location within one of the sprint nodes described in Section 4. The remaining $log_2(p) - 5$ high order bits determine the sprint node in which a processor is located. The number of hypercube links between two processors is determined by the hamming distance between the $log_2(p) - 5$ high order bits of the two addresses. . The fetching processors in Table 1 are listed in order of increasing hamming distrance from processor 0. When the hamming distance is less than or equal to 5, the time required to fetch is roughly constant. This suggests that the time required to fetch within a sprint node does not depend on the processor's location within the node. As expected, the number of router cycles was equal to the number of hypercube links, i.e. the hamming distance minus 5.

As discussed above, it is possible to transfer two floating point numbers bidirectionally, along each hypercube edge during each router cycle. We map the four nearest neighbor graph onto a two dimensional set of virtual processors, this VP set is mapped to sprint nodes so that contiguous square or rectilinear chunks of domain are mapped to each node. A binary reflected Gray code is used to embed the domain regions so that processors handling contiguous domain regions are directly connected. Since we know the number of data elements that must be exchanged along each hypercube link, we can calculate how many router cycles inter-node communication should require. Table 2 depicts the number of router cycles required and the communication time needed when this experiment was performed on 16K processors. This table also indicates the size of the largest message transmitted. In this simple case, we expect that the number of router cycles needed should be equal to half the size of the largest message. For all but the largest domain tested, this expectation proved to be accurate. The performance of the communications compiler degraded in a striking manner for the largest domain tested. The algorithm described by Dahl [6] should not exhibit this behavior, it seems likely that this degradation is due to a bug in the communications compiler software.

6.2 Relative Performance of Different Communications Mechanisms

A 16K processors Connection Machine-2 carried out a matrix vector multiply using the one Row per VP algorithm. We used a matrix generated from a four nearest neighbor graph with probabilities q equal to 0, and 0.2. The following communication mechanisms were employed:

1. Get: The standard CM-2 general router is called four times, once for each of the four off-processor data elements needed by each processor.

Table 1: **Time required for single fetch**

Fetching Processor	Time microseconds	router cycles
0	210	0
1	210	0
3	216	0
7	216	0
15	216	0
31	215	0
63	264	1
127	325	2
255	385	3
511	456	4
1023	517	5
2047	578	6
4095	629	7
8191	681	8
16383	728	9
32767	780	10

Table 2: **2-D VP set, Four nearest neighbors, 16K processors, p = 0.0**

Domain Size	Largest Message	Number of Cycles	Communication Time (ms)
128x128	8	4	0.88
128x256	8	4	1.24
256x256	16	8	2.01
256x512	16	58	4.23

Table 3: **Four Nearest Neighbor Synthetic Workload 16K processors, q = 0**

Domain	news Mflops	get Mflops	compiled get Mflops	compiled gather Mflops
128x128	164.5	10.7	73.4	154.8
256x128	198.6	12.7	99.8	180.9
256x256	253.2	15.3	144.1	226.0
512x256	295.6	16.5	159.6	230.5

2. Compiled get: Communications compiled using the communications compiler; the communications compiler preprocessor was called four times, once for each of the four off-processor data elements required by each processor. The data delivery procedure is called four times during each mesh sweep.

3. Compiled gather: Communications compiled using the communications compiler; a single call to the communications compiler preprocessor handles each processor's four data requests. For each iteration, a single data delivery function carries out all communication.

4. NEWS: CM-2 communications procedures that transmit information using mesh embedded into hypercube by binary reflected gray code. NEWS was only used to benchmark the completely uniform problem (q = 0).

In Table 3, we present timing results obtained from a completely uniform problem (q=0). The time required for the CM-2 general router ranged from 15 to 17 times that used by the NEWS network to solve the same problem. The compiled get and compiled gather were much faster, the time required for the compiled gather ranged from approximately 5% to 30% more than the NEWS network. Note that the compiled get did not produce as good results.

Table 4 depicts results obtained on 16K processors using the four nearest neighbor synthetic mesh with $q = 0.2$. While the computational rates for this problem are lower than those seen in the completely regular mesh, the communications compiler is still roughly a factor of 11 faster than the general CM-2 router. It should, however, be noted that several seconds were required for the communication compiler to schedule the routing of messages. This perprocessing time is not included in the above depicted performance measurements.

6.3 Performance Effects of Geometrically Based Mappings

We compare performance results arising from 1-D and 2-D mappings of the synthetic mesh and from meshes that are derived from aerodynamic problems. We use a range of parameters for the synthetic mesh to stress the communications network with varying degrees of irregularity. We also compare the performance we obtain when we

map aerodynamic unstructured meshes onto sprint nodes or

Table 4: **Four Nearest Neighbor Synthetic Workload 16K processors, q = 0.2**

Domain	get Mflops	compiled get Mflops	compiled gather Mflops
128x128	8.1	42.2	90.5
256x128	10.2	66.2	100.9
256x256	10.2	91.6	119.2
512x256	10.7	103.2	127.0

assign matrix rows to virtual processors in a naive fashion.

In all cases described in the section, we used the One Row per VP sparse matrix vector multiply algorithm.

6.3.1 Synthetic Workload from Templates

The performance results in Table 5 are obtained from synthetic mesh test matrices on an 8K processor CM-2. The matrices are mapped in one of the two following ways:

1. A domain's mesh points are partitioned into a two dimensional grid composed of rectangular domain regions. The two dimensional grid is Gray coded into the CM-2, with the matrix rows that correspond to each region's mesh points assigned to processors in the appropriate sprint node.

2. A domain's mesh points are ordered in a row-wise manner. The virtual processors of the CM-2 are ordered using a one dimensional Gray code, consecutively ordered rows are assigned to consecutively numbered CM-2 processors.

Consider first the q= 0 case. When the 2-D grid of virtual processors is employed, messages in the four point stencil synthetic mesh must only traverse one hypercube link, in the eight point stencil messages must traverse only two links. When we use a one dimensional grid of virtual processors, this degree of locality is not assured and we see reduced performance. As q increases, the performance obtained in the 2-D case drops dramatically, while the performance obtained in the 1-D case changes only slightly. It is noteworthy that the 1-D embedding yields substantially better performances for high q values; the reasons for this are not clear.

In Table 6 we depict, for the four nearest neighbor synthetic mesh, the time required for computation and communication, along with the number of cycles needed by the router. It is noteworthy that even when $q = 0$, 2-D VP set matrix vector multiplications are very communication intensive. For instance, when we map to a 2-D VP set, approximately 90% of the time time spent on the q= 0 case is due to communication. For $q = 0.2$, 98% of the time is attributable to communication. For $p = 0$ with a 2-D VP set, as the domain grows larger, the fraction of time spent communicating should decrease due to the decrease in the ratio of the number of data that must be communicated to the number of data that must be computed. In a 256 by 256 domain, 83% of time is spent communicating. In Table 6

Table 5: **Synthetic mesh, One Row per VP**

Probability	1-D VP array 4 pt stencil Mflops	2-D VP array 4 pt stencil Mflops	1-D VP array 8 pt stencil Mflops	2-D VP array 8 pt stencil Mflops
0.0	25.6	90.1	18.7	77.8
0.1	25.4	55.1	19.1	42.6
0.2	24.7	24.2	19.6	26.0
0.3	21.1	18.0	20.1	19.1
0.5	23.6	12.1	21.2	12.5
0.8	22.9	8.0	22.8	8.1

Table 6: **Four nearest neighbor synthetic mesh mapped to 1-D and 2-D VP sets, 16K processors**

Domain Size	q	2-D VP set One Row/VP			1-D VP set One Row/VP		
		Total Time(ms)	*Comm Time(ms)*	*Cycles*	*Total Time(ms)*	*Comm Time(ms)*	*Cycles*
128x128	0.0	0.98	0.88	4	2.62	2.82	33
128x256	0.0	1.21	1.24	4	5.23	5.17	67
256x256	0.0	2.41	2.01	8	10.18	9.48	133
128x128	0.2	4.26	4.03	58	3.03	2.75	38
128x256	0.2	7.83	8.12	110	6.46	6.38	86
256x256	0.2	14.89	14.47	207	11.27	10.10	142

we also note that increasing q from 0.0 to 0.2 does not have a large impact on the cost of communication when we use a 1-D VP set.

6.3.2 Importance of Geometrically Based Mapping - Benchmarks using Meshes from Aerodynamic Problems

In this section we look at performance results obtained from the aerodynamic unstructured meshes described in Section 3. The two aerodymanic unstructured meshes were used in iterative sparse matrix vector multiplications on 4K, 8K and 16K CM-2 configurations. The communications compiler was employed to schedule communications. We also carried out calculations to find out the number of router cycles that would be required if we were to implement a different, and simpler router algorithm.

In Table 7 we depict the total time required to carry out a sparse matrix vector multiply, the communication time and the computational rate in megaflops. We partition each mesh into a two dimensional rectilinear array of $log_2(p) - 5$ blocks using jagged partitioning. Each block is assigned to a Gray coded sprint node (this is referred to as *mapped* in Table 7. We also performed a set of experiments in which we assigned matrix row i to the i th element in a gray coded array of virtual processors. Recall that the numbering of the matrix rows is

Table 7: **Effect of Processor Assignment on Unstructured Meshes from Aerodynamics**

Mesh Type	Assignment	Number Processors	Total Time(ms)	Comm. Time(ms)	Mflop	Router Cycles
Mesh A	Mapped	4K	8.8	6.9	15.0	59
Mesh A	Naive	4K	16.8	14.9	7.9	189
Mesh A	Mapped	8K	4.8	3.8	27.4	35
Mesh A	Naive	8K	9.8	8.8	13.4	116
Mesh A	Mapped	16K	4.2	3.7	31.0	46
Mesh A	Naive	16K	9.8	9.2	23.2	131
Mesh B	Mapped	4K	8.8	6.4	26.1	32
Mesh B	Naive	4K	32.6	30.2	7.0	422
Mesh B	Mapped	8K	4.8	3.6	47.0	22
Mesh B	Naive	8K	18.4	17.1	12.4	244
Mesh B	Mapped	16K	5.7	5.1	39.8	59
Mesh B	Naive	16K	9.8	9.2	23.2	131

not closely tied to geometric locality. This is referred to as *naive* in Table 7.

For 4K and 8K processors, we see a difference of roughly a factor of 2 to 4 between the computational rates achieved by the mapped and the naive processor assignment. We see only a modest improvement in total time in Mesh A when we go from 8K to 16K processors. Recall that the number of mesh rows is 11,143 which is smaller than the number of processors (although the number of mesh rows is still much larger than the number of hypercube nodes). In these matrices, different rows have different numbers of non-zero elements. Table 7.

In Table 7, we depict the number of router cycles required by the communications compiler to carry out the communication involved in each problem. It is not surprising that the mapped processor assignments required many fewer router cycles than did the naive processor assigments. It is notable that when we performed mapped assignment for both Mesh A and Mesh B, the number of router cycles was minimized when we used 8K processors. This result is not surprising since for a fixed size problem, increasing machine size both increases the potential communication concurrency, it can also increase the distance over which data must be routed.

6.4 A comparison between sparse matrix vector multiply algorithms

We compare the One Row per VP and the Reduction sparse matrix vector multiply algorithm on a 16K processor CM-2 using the four nearest neighbor synthetic mesh (K=4 with $p = 0.0$ and 0.2). We use the four nearest neighbor synthetic mesh to generate an $N * M$ row matrix. When we use the One Row per VP algorithm to carry out the sparse matrix vector multiply, we assign the sparse rows of the matrix to a size $M * N$ one dimensional logical array of virtual processors. When we use the Reduction algorithm, we declare a C_R by $M * N$ grid of virtual processors and assign each element of the matrix to a separate virtual processor. As

Table 8: One Row Per VP v.s. Reduction 1-D VP set, Four Nearest Neighbors

Domain	p = 0.0		p = 0.2	
	one row/vp Mflops	reduction Mflops	one row/vp Mflops	reduction Mflops
64x64	12.9	24.4	11.5	21.1
64x128	25.8	35.2	19.9	27.8
128x128	51.6	41.5	44.6	32.1
256x128	52.5	41.7	38.9	34.4
256x256	53.0	42.7	43.2	41.2

described in Section 4, this ensures that the low order virtual processor dimension (length C_R) is mapped to low order address bits; i.e., reduction is sequentialized when the problem is large enough to fit a row onto each physical processor.

In Table 8, we note that the Reduction algorithm performs better than the One Row per VP algorithm in matrices arising from 64x64 point and 64x128 point domains. In these matrices, there is fewer than one virtual processor per physical processor. In these cases, the extra parallelism afforded by the Reduction algorithm yields benefits. For domains size 128x128 and above, the inter-processor and inter-hypercube communication patterns are both identical, and in both algorithms reduction is sequentialized. While the One Row per VP algorithm performs slightly better than the Reduction algorithm for large domains, the difference can be traced to difference in the efficiencies achieved by the different PARIS instructions used in the algorithms.

When we recall that it is really the floating point chips that perform calculations, we will realize the the above discussion is a bit misleading. These results underline some of the disadvantages inherent in an architecture that couples bit serial processors to bit parallel floating point chips. On a 64x64 domain, in the One Row per VP algorithm, we assign 8 rows to each sprint node. Nevertheless, when we assign fewer than one row per bit serial processor, we see a striking performance degradation. The relationship between performance and the ratio of virtual to physical processors (VP ratio) is well known ([18]).

In Table 9 we compare the results obtained using the One Row per VP and the Reduction algorithms for banded matrices with N rows and bandwidths C = 4,8 and 16. The behavior as a function of C is complex as it is determined by tradeoffs between:

- ratio between number of computations and communications
- distance transmitted information must travel
- ratio between virtual and physical processors (in the reduction cases)

Note that we include timings not only from the standard Reduction algorithm but from the Reduction algorithm with Replication. Except for the N = 4K, C = 16 case, One Row per VP performs better than Reduction.

Reduction with Replication consistently out-performs Reduction. If the CM-2 were actually using bit serial processors to perform computations and communications, this discrepancy would be easy to explain. In the CM-2 however, a good explanation of this difference

Table 9: **Comparison between One Row per VP and Reduction Algorithms for Banded Matrices**

		Reduction		**Repl. Reduction**		**One Row / VP**	
N	*C*	*Time(ms)*	*Mflops*	*Time(ms)*	Mflops	*Time(ms)*	*Mflops*
64 K	8	8.1	129	5.5	189	4.17	252
	4	4.4	120	2.8	180	2.0	264
16 K	16	5.3	99	3.9	132	4.2	126
	8	2.3	112	1.7	147	1.5	180
	4	1.4	90	1.0	131	0.8	159
4 K	16	2.5	52	2.1	62	3.3	39
	8	1.4	48	1.01	65	1.3	50
	4	1.0	32	0.70	47	1.0	34

would have to take into account some fairly subtle interactions between the algorithm, the communications compiler, the way in which PARIS supports virtual processors and the hardware.

7 Conclusion

A number of researchers have used the CM-2 to solve sparse and irregular problems. Groups who used the relatively expensive general router encountered mixed results. Calculations arising from sparse linear algebra seemed to fare poorly; the router operations were amortized by relatively few computations [3], [17]. Other computations had a higher ratio of computations to routing operations, and the general router proved to be less of a bottleneck [5].

Several researchers have proposed methods to optimize the performance of inter-processor communication arising from repeatedly used irregular data access patterns. These efforts have been directed towards the MPP [7], the Intel iPSC multiprocessors [22] as well as towards the CM-2 [6]. Hammond [12] reported results using the communication compiler on an unstructured mesh problem arising in computational fluid dynamics. We have used the sparse matrix vector multiply kernel to carry out a detailed study of the performance effects of irregular communications patterns on the CM-2.

We carried out a set of experiments to characterize the communications capabilities of the CM-2 under a variety of controlled conditions. The efficiencies achieved are critically dependent on the communications pattern. In sparse matrix vector multiplication, the communications pattern is dependent on the structure of the sparse matrix, as well as on the way in which the problem is mapped onto the machine.

In the process of carrying out our performance evaluation, we developed and then made extensive use

a parameterized synthetic mesh,

unstructured meshes that had been generated for aerodynamic codes, and a set of sparse matrices with banded patterns of non-zeros.

This benchmarking suite stressed the communications capabilities of the CM-2 in a range of different ways.

Even on well partitioned problems, use of the general router led to order of magnitude performance degradations (Section 6.2). While poor mapping and problem irregularity did lead to performance degradation, the degree of degradation was typically more modest (Section 6.3). Much work has been focused on developing effective problem partitioning methods (e.g., [2], [9], [10]). The partitioning methods typically attempt to evenly partition computational work while reducing some objective function related to interprocessor communication costs. The results presented in this paper (along with those described in [12]) make it clear that, in the absence of mechanisms to ensure effective communications network utilization, careful problem partitioning may have only a very limited performance impact. On the other hand, particularly when used in conjunction with good methods for partitioning problems, important payoffs can come from developing mechanisms to increase the utilization of communications networks.

The results presented here suggest that the performance of sparse and unstructured computations on the CM-2 can be optimized to a considerable extent. There is reason to pursue investigations into improved communication compiler algorithms. We have carried out preliminary experimentation which indicates that it may be possible to further speed communications through better algorithms to assign messages to CM-2 communications links. It also would seem likely that the preprocessing costs could be substantially reduced.

8 Acknowlegements

We would like to thank Harry Jordan for his careful editing of this manuscript and to Steve Hammond and Denning Dahl for helpful discussions.

References

[1] M. Berger and S. H. Bokhari, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Trans. on Computers, C-36 (1987), pp. 570–580.

[2] M. J. Berger and S. H. Bokhari, *A partitioning strategy for pdes across multiprocessors*, in The Proceedings of the 1985 International Conference on Parallel Processing, August 1985.

[3] H. Berryman, J. Saltz, and W. Gropp, *Krylov methods with incomplete factorization preconditioners on the cm-2*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 186–190.

[4] S. Borkar, R. Cohn, G. Cox et al., *iwarp: An integrated solution to high-speed parallel computing*, in Proc. of the Supercomputing Conference, November 1988, pp. 330–339.

[5] J. Cerutti and H. Trease, *Free lagrange methods on the connection machine*, in To appear: Proceedings of the 2nd International Conference on Free Lagrange Methods, Springer-Verlag, 1991.

[6] E. D. Dahl, *Mapping and compiled communication on the connection machine system*, in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston S.C., 1990.

[7] J. Dorband, *Sort computation*, in Proceedings of the 2nd Symposium on the Frontiers of Massively Parallel Computation, October 1988, pp. 137–141.

[8] I. S. Duff and J. K. Reid, *Direct Methods for Sparse Matrices*, Oxford Science Publications, Oxford University Press, New York, 1986.

[9] G. Fox, *A graphical approach to load balancing and sparse matrix vector multiplication on the hypercube*, in The IMA Volumes in Mathematics and its Applications. Volume 13: Numerical Algorithms for Modern Parallel Computer Architectures Martin Schultz Editor, Springer-Verlag, 1988.

[10] G. Fox, A. Kowala, and R. Williams, *The implementation of a dynamic load balancer*, in The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN, 1987, pp. 114–121.

[11] *Numerical methods for the computation of inviscid transonic flows with shock waves - a gamm workshop*, in Notes on Numercial Fluid Mechanics, Vol. 3.

[12] S. Hammond and R. Schreiber, *Mapping unstructured grid problems to the connection machine*, Report 90.22, RIACS, October 1990.

[13] D. J. Mavriplis, *Multigrid solution of the two-dimensional Euler equations on unstructured triangular meshes*, AIAA Journal, 26 (1988), pp. 824–831.

[14] M. Misra and P. Kumar, *Efficient vlsi implementation of iterative solutions to sparse linear systems*, Report 246, Institute for Robotics and Intelligent Systems, University of Southern California, 1988.

[15] *Using the connection machine system (paris), Volume 3*, Report ANL/MCS-TM-118, Argonne National Laboratory, June 1989.

[16] S. Petiton, *Parallel qr algorithms for iterative subspace methods on the connection machine*, in Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing, December 1989.

[17] B. Philippe and Y. Saad, *Solving large sparse eiginvalue problems on supercomputers*, in Parallel and Distributed Algorithms, M. C. et al., ed., North Holland, 1989.

[18] R. Pozo and A. MacDonald, *Performance characteristics of scientific computation on the connection machine*, Report 440, Computer Science Dept, University Colorado at Boulder, 1989.

[19] J. Ratner, *Personal communication*, tech. rep.

[20] Y. Saad, *Sparsekit: a basic tool kit for sparse matrix computations*, Report 90-20, RIACS, 1990.

[21] J. Saltz, H. Berryman, and J. Wu, *Runtime compilation for multiprocessors*, Report 90-59, ICASE, 1990.

[22] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman, *Run-time scheduling and execution of loops on message passing machines*, Journal of Parallel and Distributed Computing, 8 (1990), pp. 303–312.

[23] P. Venkatakrishnan, *Preconditioned conjugate gradient methods for the compressible navier stokes equations*, to appear - AIAA Journal, (1991).

[24] P. Venkatkrishnan, J. Saltz, and D. Mavriplis, *Parallel preconditioned iterative methods for the compressible navier stokes equations*, in 12th International Conference on Numerical Methods in Fluid Dynamics, Oxford, England, July 1990.

[25] D. L. Whitaker, D. C. Slack, and R. W. Walters, *Solution algorithms for the two-dimensional euler equations on unstructured meshes*, in Proceedings AIAA 28th Aerospace Sciences Meeting, Reno, Nevada, January 1990.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-187514 ICASE Report No. 91-12	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle PERFORMANCE EFFECTS OF IRREGULAR COMMUNICATIONS ON MASSIVELY PARALLEL MULTIPROCESSORS		5. Report Date January 1991
		6. Performing Organization Code
7. Author(s) Joel Saltz, Adam Rifkin, Serge Petiton, Harry Berryman		8. Performing Organization Report No. 91-12
		10. Work Unit No. 505-90-52-01
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		11. Contract or Grant No. NAS1-18605
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Michael F. Card

Submitted to Journal of Parallel and Distributed Computing

Final Report

16. Abstract

We conduct a detailed study of the performance effects of irregular communications patterns on the CM-2. We characterize the communications capabilities of the CM-2 under a variety of controlled conditions.

In the process of carrying out our performance evaluation, we develop and make extensive use of a parameterized synthetic mesh. In addition we carry out timings with unstructured meshes generated for aerodynamic codes and a set of sparse matrices with banded patterns on non-zeros. This benchmarding suite stresses the communications capabilities of the CM-2 in a range of different ways. Our benchmark results demonstrate that is is possible to make effective use of much of the massive concurrency available in the communications network.

17. Key Words (Suggested by Author(s)) SIMD, sparse computation, matrix vector multiply, communications compiler, connection machine, CM-2	18. Distribution Statement 60 - Computer Operations and Hardware 61 - Computer Programming and Software Unclassified - Unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of pages 22	22. Price A03